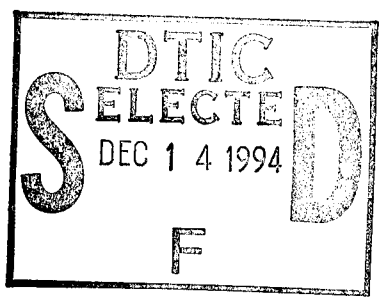
AEROSPACE REPORT NO.
ATR-92(2778)-7

A Proposal for the Verification in SDVS of a Portion of the MSX Tracking Processor Software

30 September 1992



Prepared by

T. K. MENAS
Computer Systems Division

Prepared for

NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

Engineering and Technology Group

Aerospace Report No.
ATR-92(2778)-7

A PROPOSAL FOR THE VERIFICATION IN SDVS OF A PORTION OF THE MSX TRACKING PROCESSOR SOFTWARE

Prepared by

T. K. Menas
Computer Systems Division

30 September 1992

Engineering and Technology Group
THE AEROSPACE CORPORATION
El Segundo, CA 90245-4691

Prepared for

NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

Report No.
ATR-92(2778)-7

A PROPOSAL FOR THE VERIFICATION IN SDVS OF A PORTION OF THE MSX TRACKING PROCESSOR SOFTWARE

Prepared

T. K. Menas

Approved

B. H. Levy, Manager
Computer Assurance Section

D. B. Baker, Director
Trusted Computer Systems Department

C. A. Sunshine, Principal Director
Computer Science and Technology Subdivision

Abstract

The Midcourse Space Experiment (MSX) is a Strategic Defense Initiative Organization program whose primary purpose is to conduct tracking event experiments of targets/phenomena in midcourse.

In this report we describe the portion of the MSX spacecraft tracking processor software that we have selected for verification in SDVS. We then enumerate the Ada constructs appearing in this part of the software that are not currently handled by the SDVS Ada translator, but which we intend to implement. We also mention some of the problems that we expect to encounter in the course of the project.

Contents

Acknowledgments

We are especially indebted to Richard Waddell and Shane Hutton of the Johns Hopkins University Applied Physics Laboratory for their help in discussions of issues raised in this report. We are also grateful for the help given to us by Mark Bouler, Ivan Filippenko, Beth Levy, David Martin, and Leo Marcus of The Aerospace Corporation.

1 Introduction

The purpose of this report is to document the first phase of a joint project between The Aerospace Corporation (Aerospace) and the Johns Hopkins University Applied Physics Laboratory (JHU/APL) to verify a portion of the Midcourse Space Experiment (MSX) spacecraft tracking processor software. Primarily, the purpose of the project is to evaluate the applicability of the State Delta Verification System (SDVS) to the verification of application programs written in high-level languages. A byproduct will be the knowledge gained by JHU/APL of the value of applying formal methods in DoD programs.

The first phase of the project consisted of

(i) selecting and analyzing a portion of the MSX Ada code to be verified,

(ii) examining the documentation pertaining to that portion, and

(iii) delineating the Ada constructs that appear in that portion but that the SDVS Ada translator does not currently handle. (For a description of SDVS see [1], [2], and [3].)

MSX is a near-term Strategic Defense Initiative Organization program whose primary purpose is to conduct tracking event experiments of targets/phenomena in midcourse. In a tracking event, the main spacecraft systems are the command processor, the attitude processor, the tracking processor, the sensors, and the data-handling system (telemetry). The command processor receives, buffers, and relays commands for a network consisting of the ground and the spacecraft subsystems. The attitude processor interfaces to the attitude sensors and controllers. Its primary function is to determine and control the spacecraft attitude. The fundamental function of the tracking processor is to generate sufficient information for the attitude processor to point the spacecraft at the desired target, location, or direction. The tracking processor is designed around a MIL-STD-1750A (1750A) microprocessor with 2K of ROM, 512K of RAM, and 256K of EEPROM.

When the spacecraft is not involved in a tracking event, the tracking processor is turned off to conserve power. During these periods, the direction of the spacecraft is controlled autonomously by the attitude processor and is in "parked mode." Before a tracking event is to take place, the tracking processor is turned on by a real-time or delayed command. Then the ROM is used for power up, the software and data for the tracking event are loaded into storage from EEPROM to RAM, and the event begins.

During the current event and for the preparation of the next tracking event, commands are generally uplinked from the ground to the command processor and relayed to the tracking processor via a serial port. These serial digital commands are combined by the tracking processor to form application-level messages, which are then stored in RAM, EEPROM, or both. There are eleven types of application-level messages. One of these, the data-structure memory-load application message (data-structure message, for short), can modify up to about 120 tracking parameters used in a tracking event. The number of commands required to form a data-structure message is a function of the type of tracking parameter the data-structure message modifies.

The *target software* we have selected for verification is that part of the tracking processor software that processes serial digital commands from the command processor into data-structure messages and then stores the messages into RAM, EEPROM, or into both. Our knowledge of the MSX program in general, and of the specifications and the software for the tracking processor in particular, was culled from discussions with Richard Waddell and Shane Hutton of JHU/APL and from the following documents:

- R. L. Waddell, "MSX Tracking Processor Software Requirements Specification," S1A-104-89 JHU/APL, 1989;
- R. L. Waddell and S. Hutton, "Midcourse Space Experiment (MSX) Tracking Processor Software Functional Design," S1A-031-90 JHU/APL, 1990;
- S. F. Hutton, "Tracking Processor / Command Processor Interface Design Specification (Version 2)," S1A-136-91 JHU/APL, 1991;
- G. Heyler, S. Hutton, and R. Waddell, "Midcourse Space Experiment (MSX) Tracking Processor Software Detailed Design Document," S1A-084-91 JHU/APL, 1991;
- a portion of the MSX software.

The specification for the target software will be extracted primarily from the third of the documents listed above: this document contains the specifications for all of the application messages and data structures loaded from the command processor to the tracking processor.

In Section 2 of this report we give a brief outline and description of the target software; some of the code will probably not be translated by the SDVS Ada translator, for reasons we will make clear. We intend to characterize these parts of the code by either state deltas or by a form of the SDVS Ada offline characterization facility [4].

Section 3 is an enumeration of the Ada constructs [5] in the target software that the SDVS Ada translator does not currently handle. For each such construct, we will discuss our proposed solution. The possible solutions are as follows:

- implement the construct in SDVS
- delete[1] it from the MSX example
- replace it by functionally equivalent Ada code
- replace it by nonequivalent Ada code (tasking)
- encapsulate procedure and function bodies in which it occurs by means of the SDVS offline characterization facility

Finally, in the last section, we discuss our current work on the project and our plans for its near future.

[1] All deletions and replacements will be made in the software that will be translated by the SDVS Ada translator and will not affect the actual software for the MSX program.

2 A Functional Overview of the Code

We focus in this section on that part of the tracking processor software that receives and buffers up to 70 commands from the command processor, builds the appropriate application messages from a list of commands, stores these messages in a circular message queue, and then processes the messages according to their specifications. During the construction of the messages, a number of tests are performed on the integrity of the commands and messages, and the status of the tracking processor is reported to telemetry.

Our intent is to verify that the data-structure messages are built according to their specification. To illustrate the type of serial digital commands required to build a data-structure message and to get an overview of its construction, consider the Beacon Alignment First Object data-structure message. This message encodes a 3×3 real matrix that is required to be stored in both EEPROM and RAM. The matrix has 9 real number entries, and each real number requires 4 bytes. Therefore, the matrix requires a total of 36 bytes of data.

A byte is 8 bits long; a word is 16 bits long; and a command consists of two words. In the Ada code, bytes and words are represented by integers constrained to specific ranges. Although bytes (words) have an integer parent type, they encode sequences of 0's and 1's that are 8 (16) bits long. For example, if the byte $B = 7$, B encodes (i.e., contains) the bit sequence $< 00000111 >$.

Fourteen commands are needed for the construction of the Beacon Alignment First Object data structure (see Table 1). The first bit of the first byte of each command is the parity bit for the entire command. The other bits serve the functions we outline below.

(i) Command 1:

- The last seven bits of the first byte encode the op-code of the command. For a command that begins a data structure load message, the op-code is 1.
- The second byte encodes the storage information: EEPROM, RAM, or both. For the Beacon Alignment First Object data structure, which according to the specifications must be stored in both EEPROM and RAM, this byte must be equal to 2.
- The third byte contains the identification code, ID, of the data structure. For the Beacon First Alignment Object, this code is 1.
- The fourth byte is the first byte of data, d_1, for the matrix.

(ii) Command n where $1 < n \leq 12$:

- The last seven bits of the first byte encode the op-code, which is 8, for a data-structure load continuation command.
- The other three bytes are the final bytes of data for the matrix: d_{3n-4}, d_{3n-3}, and d_{3n-2}.

(iii) Command 13:

- The last seven bits of the first byte encode the continuation op-code 8.
- The next two bytes are the next bytes of data for the matrix: d_{35} and d_{36}.

- The fourth byte is the first byte of the 2-byte checksum.

(iv) Command 14:

- The last seven bits of the first byte encode the continuation op-code 8.
- The second byte is the second byte of the checksum.
- The third and fourth bytes are spares.

Table 1: Beacon Alignment First Object

	Byte 1		Byte 2	Byte 3	Byte 4
Command 1	P[2]	1	2	1	d_1
Command 2	P	8	d_2	d_3	d_4
Command n	P	8	d_{3n-4}	d_{3n-3}	d_{3n-2}
Command 13	P	8	d_{35}	d_{36}	checksum
Command 14	P	8	checksum	spare	spare

[2] P is the parity bit.

2.1 Process Commands/Data from Command System

We have examined eighteen library units that fall either in the area or in the periphery of our target software. Below, we group these units according to their function, list some of their more important subunits, and discuss their role, if any, in our target software. Units: ARRAY_OF_BLOCKS, SERIAL_DIG_COMMANDS, and APP_MSGS

(i) ARRAY_OF_BLOCKS
ARRAY_OF_BLOCKS is used only for long-memory loads, which are not in the category of application messages we have selected for verification; we will thus not consider this package, but will describe the remaining two in some detail.

(ii) SERIAL_DIG_COMMANDS
This Ada package contains the procedures CMD_IN_HANDLER and RET_CMD_BUF_AND_STATUS.

- CMD_IN_HANDLER is a procedure that services interrupts that occur when the command processor is ready to transmit a command to the tracking processor. It obtains the command in the format of an array of 4 words (see Table 2), processes the array as a packed record of 4 byte fields (see Table 2), and then stores this command/record in a buffer that can buffer up to 70 commands (see Table 3). It stores the status of each command in a separate status buffer and also performs parity checks.

Table 2: Command Formats

Command Obtained: Array of 4 Words

00000000	First Byte
00000000	Second Byte
00000000	Third Byte
00000000	Fourth Byte

Command Processed: Packed Record

First Byte	Second Byte
Third Byte	Fourth Byte

Table 3: Cmd_Buf

Command 1	First Byte	Second Byte
	Third Byte	Fourth Byte
Command 2	First Byte	Second Byte
	Third Byte	Fourth Byte
Command 3	First Byte	Second Byte
	Third Byte	Fourth Byte
	⋮	⋮
Command N	First Byte	Second Byte
	Third Byte	Fourth Byte

- The procedure RET_CMD_BUF_AND_STATUS is called within an infinite loop by a task in APP_MSGS to retrieve the command and command-status buffers created by CMD_IN_HANDLER.

(iii) APP_MSGS

This Ada package contains the tasks BUILD, PROCESS_MSG, and MANAGE_MSG_RETRIEVAL, and the two procedures INITIATE_BUILD and CONTINUE_BUILD.

- BUILD calls SERIAL_DIG_COMMANDS.RET_CMD_COUNT repeatedly to see if there are any commands in the command buffer to be processed into messages. If there are, it then calls SERIAL_DIG_COMMANDS.RET_CMD_BUF_AND_STATUS to obtain the command buffer. If the op-code of the first command in the command buffer indicates that this command is the beginning of an application message, BUILD calls INITIATE_BUILD to start the build of the message. Otherwise, it calls CONTINUE_BUILD to continue the build of the application message currently being processed. CONTINUE_BUILD may be called repeatedly until the message is complete (see Tables 4 and 5). Note that the message is constructed in stages; it is quite possible BUILD will retrieve more than one command buffer to construct a message. When a message has been successfully constructed, CONTINUE_BUILD places the message in a circular message queue (see Table 5) that may hold up to 30 messages and control returns to BUILD. If BUILD has constructed at least one message from the most recently retrieved buffer of commands, it waits for a rendezvous with the task MANAGE_MSG_RETRIEVAL.

- If the message queue is not empty, PROCESS_MSG will rendezvous with the task MANAGE_MSG_RETRIEVAL to retrieve the message at the head of the queue. If the message is a data structure load, it writes it in EEPROM or RAM or in both (depending on the information contained in the message).
- The task MANAGE_MSG_RETRIEVAL coordinates the execution of the other two tasks by a guarded select statement. It shares a message-counter variable with CONTINUE_BUILD; CONTINUE_BUILD increments this variable when it stores a message in the message queue and MANAGE_MSG_RETRIEVAL decrements it upon its rendezvous with PROCESS_MSG. This message-counter variable must be positive for a rendezvous to take place between MANAGE_MSG_RETRIEVAL and PROCESS_MSG.

After Initiate_Build

1	0	0
2	0	OP
3	EEPROM/RAM	ID
4	d_1	To Be Continued

After Continue_Build 1st Time

1	0	0
2	0	OP
3	EEPROM/RAM	ID
4	d_1	d_2
5	d_3	d_4
6	To Be Continued	

After Continue_Build 2nd Time

1	0	0
2	0	OP
3	EEPROM/RAM	ID
4	d_1	d_2
5	d_3	d_4
6	d_5	d_6
7	d_7	To Be Continued

After Continue_Build 3rd Time

1	0	0
2	0	OP
3	EEPROM/RAM	ID
4	d_1	d_2
5	d_3	d_4
6	d_5	d_6
7	d_7	d_8
8	d_9	d_{10}
9	To Be Continued	

Table 4: Piecewise Building of Application Message

2.2 Ada Packages that Depend on the Tartan Compiler

Units: INTRINSICS, INTRINSIC_FUNCTIONS, and UNCHECKED_CONVERSION

The INTRINSICS package is a part of the Tartan[3] library of packages and contains generic functions and procedures that are system dependent. They are generally used to do bit

[3]The MSX Tracking Processor Ada software will be compiled by a Tartan ([6]) compiler.

Completed Message

1	0	0
2	0	OP
3	EEPROM/RAM	ID
4	d_1	d_2
5	d_3	d_4
⋮		
n	$d_{2(n-4)+1}$	$d_{2(n-4)+2}$
⋮		
21	d_{35}	d_{36}
22	checksum	checksum

In App_Msg_Q

1	0	OP
2	EEPROM/RAM	ID
3	d_1	d_2
4	d_3	d_4
⋮		
n	$d_{2(n-3)+1}$	$d_{2(n-3)+2}$
⋮		
20	d_{35}	d_{36}
21	checksum	checksum

Table 5: Completion and Storage of Message

manipulations, such as the masking of bits. The INTRINSIC_FUNCTIONS package contains instantiations of the generic functions and procedures in the INTRINSICS package. The UNCHECKED_CONVERSION generic function is used to convert between different data types. Since these packages are used extensively in the target software, and since their implementation is compiler-dependent, we will encapsulate the functions and procedures that they contain by using the SDVS Ada offline characterization utility.

2.3 Type Definitions

Units: GLOBAL_TYPES and CMDS_TYPES

As their names indicate, these two packages contain the type definitions (such as WORD and BYTE, which are derived integer types) common to most of the code.

2.4 Output Telemetry

Units: TM, TM_DATA, and PRIME_SCIENCE_DATA

All three of these Ada packages appear only peripherally in our target software: calls to subprograms or rendezvous with tasks of these packages are made only to report housekeeping data for telemetry (such as system errors). For this reason, it should be relatively easy to bypass them in the translation of the target software.

2.5 Tracking Parameters

Units: TRKING_DATA_STRUC and TRKING_PARAMS

The TRKING_DATA_STRUC package in this group is only used in the target software to determine the number of words needed to build each type of data-structure application message; therefore, the package will be translated into SDVS (or at least the portions of the package that are needed). The second package in this group, TRKING_PARAMS, is only used by the task PROCESS_MSG of our target software to store a tracking-structure application message by an extended rendezvous with the task MANAGE of TRKING_PARAMS for those data structures to be stored in RAM (or in both RAM and EEPROM). The rendezvous only passes the data structure to be stored and could be easily specified in SDVS by a form of offline characterization.

2.6 Control Mode/State

Unit: MODE_STATE

The Ada package MODE_STATE is only used in our target software to obtain information about the status of the mode state, e.g., power_up, init_EEPROM_write, tracking. It should be easy to translate the results of such calls by offline characterization.

2.7 Track Targets

Units: POINTING_INFO_OUT and TRK_TARGETS_ISR_PKG

TRK_TARGETS_ISR_PKG never appears in the target software, and POINTING_INFO_OUT is used by PROCESS_MSG to process a message that is not a data structure load.

2.8 Miscellaneous Packages

Units: EEPROM_DATA, ASM_UTILITY, and MEMORY_MANAGER

All three Ada packages have package bodies written in 1750A assembly code. Portions of our target software do in fact call procedures or functions that appear in these packages, but the packages will not be translated by the SDVS Ada translator. When required, these procedure/function calls will be encapsulated by a form of offline characterization.

3 Possible SDVS Enhancements

There are many Ada constructs in the target software that are not currently "compilable" by the SDVS Ada translator. (For brevity we will henceforth refer to these as *targeted constructs*). Of these, the most intractable are tasking and Ada features that pertain to it, eg., *delay* statements and the *priority* pragma. Below, we list each targeted construct and note our proposed solution to its presence in the target software.

(i) LONG_FLOAT and LONG_INTEGER types
These types are not part of standard Ada and are compiler dependent. We will declare LONG_FLOAT (LONG_INTEGER) to be of type FLOAT (INTEGER).

(ii) WRITESTRING and WRITELN
These are not in the Language Reference Manual [5] and appear in the Ada code only temporarily (for testing purposes). We will delete them from the target software.

(iii) *With* clause
Currently, SDVS allows this clause only for the standard INTEGER_IO and TEXT_IO packages. Furthermore, the only Ada unit that may be translated into the state delta language by the use of *adatr* in SDVS is a main Ada procedure. We have already started work on the *adatr* implementation, which will involve the addition of the capability in SDVS to *adatr* packages.

(iv) Integer subtypes
Although there is only one instance of an integer subtype definition in the target software, we think that integer subtype definitions are so prevalent in Ada programs in general that it behooves us to implement them.

(v) Integer definition derived types
These type definitions appear in several important parts of the target software (WORD and BYTE are defined in this manner). We will implement integer derived type definitions.

(vi) Type conversions
We will implement them.

(vii) Array initialization using (others =>)
We will implement it.

(viii) Two-dimensional arrays
SDVS currently handles only one-dimensional arrays. At this point, we intend to "Curry" the two-dimensional arrays, that is, to rewrite the code in such a way so that a two-dimensional array is represented as a one-dimensional array of arrays. This solution is simple enough; if time permits, we will consider the implementation of multidimensional arrays.

(ix) Named parameters
We will change instances of named parameter notation to their positional parameter equivalent in the target software.

(x) Hexadecimal notation
There are many instances of this in the target software; since they are all easy to convert to base 10 notation, we will do so (rewrite the instances).

(xi) UNCHECKED_CONVERSION
This generic function is instantiated many times in the target software and its instantiations are used repeatedly and in situations that are compiler dependent, e.g., to convert from one object to another of smaller size. We are still undecided on its exact implementation. It appears that we will not handle all situations uniformly. Certain cases are obvious, but others are not.

(xii) *Size* representation attribute
Here is an important instance in the code:

```
type WORD is range 0..65535; for WORD'size use 16;
```

In fact, all instances in the Ada code are precisely of this form: integer derived types whose size attribute matches their range. We propose to implement the size attribute so as to allow precisely these cases. It should be noted that if an object has a constrained integer type declaration, then SDVS will ensure that any assignment to that object obeys the restriction.

(xiii) *Address* representation attribute
This attribute is used either to assign to or store values of an Ada object by means of Ada procedures that have 1750A assembly code bodies. For example, in the procedure CMD_IN_HANDLER, the array of four words, Cmd_Unpacked, is assigned a value by means of the following call to the procedure READ_FIFO of the package MEMORY_MANAGER (the body of READ_FIFO is in 1750A assembly):

```
MEMORY_MANAGER.READ_FIFO(Cmd_In_FIFO_Addr, Cmd_Unpacked'address,
                         Cmd_Size);
```

In one instance, the size attribute is used not only to assign a value to an Ada object, but to do a type conversion as well. All such instances in the target software will be replaced by simple Ada procedures that will mimic the result of calls to the original procedures.

(xiv) Pragmas *elaborate*, *priority*, *pack*, *Foreign_Body*, and *Linkage_Name*
Ada pragmas are instructions to the compiler and are generally not meant to change the semantics of an Ada program. The last two pragmas in the above list are peculiar to the Tartan compiler and are used primarily for the interfaces to the assembly code, which is not a part of the target software we will translate into SDVS; we therefore think that we may safely delete these pragmas from the target software.

The first pragma, *elaborate*, may be deleted if we compile the relevant units in the correct order in SDVS. In this case, any proof of correctness would be a proof under the assumption that the packages are compiled in the correct order. Since this is not entirely satisfactory, we will implement *elaborate* if time permits.

The second pragma, *priority*, may guide the way we handle the tasking in the SDVS translation of the code, but we will not implement it, since we will not implement tasking for this project.

The third pragma, *pack*, is used for a particular format in the representation of data in the target software, but we do not think its use has any import *for the part of the code we will consider.* We think its importance lies in that part of the Ada code that interfaces with parts written in the 1750A assembly language and therefore have decided to delete it from the target software. The reader will note that none of the pragmas, with the possible exception of *elaborate*, will be directly implemented in SDVS.

(x) Tasks

As we have already indicated, the most important and extensive part of the target software we intend to verify, the part of APP_MSGS that builds and processes the application messages from the buffered serial-digital commands, contains three tasks that basically do all of the work. Tasking is perhaps the most difficult of the Ada features for an operational verification system such as SDVS to handle efficiently. The reason for this is that, in SDVS, the correctness of an Ada program is proved by symbolically executing the SDVS translation of the program. Because of this feature of SDVS, a branch in a program may necessitate the execution to completion of all the branches. If a program consists of several tasks that are executed concurrently, then every step in the execution of the program may, in effect, be a branch: the next statement to be executed may be any of the next statements in the tasks. These possibilities lead to an exponential growth in the number of possible program executions.

Currently, our solution to the tasking problem is to translate the tasks as procedures and to treat rendezvous as procedure calls. A main procedure will stipulate the order of execution of the tasks in question. In effect, we will consider the correctness of only one of the myriad possible program executions. The layout of the main procedure will be influenced, but not solely determined, by the *priority* pragma and the *delay* statement in APP_MSGS as well as by the rendezvous that must take place to complete a linear execution of the construction and processing of an application message.

4 Current and Future Work

We are now defining the semantics of the *with* clause and will proceed with its implementation and with the implementation of the other targeted constructs that we have decided to add to the SDVS Ada translator. Also, we have started to work on the SDVS specifications of the targeted Ada software.

Since the implementation of the targeted constructs will require a great deal of time to complete, we propose, as a first attempt, to rewrite[4] the portions of the code in which they appear and to translate and execute the revised code in the current SDVS 12 version. This endeavor should at least indicate problems that may arise in the next phases of the verification project. For example, the abstract-syntax Ada trees created by the SDVS translator may be inordinately large for the system to handle: over 1,000 lines of undocumented Ada code must be translated by the SDVS translator, and there are scores of object and object-type declarations in these lines that may mire the system during the symbolic execution.

The most serious problem we will encounter is certain to be the formulation of a workable specification that accounts for the many possibilities that may arise in the execution of the Ada software. These possiblities are a result of the Ada tasking used in the software. As we pointed out in the last section, SDVS does not currently handle Ada tasking and will probably not do so in the near future given that the verification of concurrent programs presents serious difficulties.

[4] We will restore the targeted constructs in the modified code to their original version, after we implement them in the SDVS Ada translator. But many of the modifications we will make to the original target software will remain in the final version of the program whose correctness we hope to prove (procedures for tasks, for example). Some of these modifications will not alter the functionality of the original code, but others will. In the final analysis, we will not prove the correctness of the original code. However, we think that any errors discovered in the modified code will be a result of errors in the original code.

References

[1] J. V. Cook, I. V. Filippenko, B. H. Levy, L. G. Marcus, and T. K. Menas, "Formal Computer Verification in the State Delta Verification System (SDVS)," in *Proceedings of the AIAA Computing in Aerospace Conference*, (Baltimore, Maryland), pp. 77–87, American Institute of Aeronautics and Astronautics, October 1991.

[2] L. G. Marcus, "SDVS 11 Users' Manual," Technical Report ATR-92(2778)-8, The Aerospace Corporation, September 1992.

[3] T. K. Menas, J. V. Cook, I. V. Filippenko, B. H. Levy, and L. G. Marcus, "SDVS 10 Tutorial," Technical Report ATR-91(6778)-11, The Aerospace Corporation, September 1991.

[4] J. E. Doner and J. V. Cook, "Offline Characterization of Procedures in the State Delta Verification System (SDVS)," Technical Report ATR-90(8590)-5, The Aerospace Corporation, September 1990.

[5] U. S. Department of Defense, *Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A)*, 22 January 1983.

[6] Tartan Laboratories Inc., *Tartan Ada VMS 1750A Compilation System Version 3.1*, December 1990.